Disentangling the Octopus: Towards Decomposing Reinforcement Learning Systems in to Intuitive Subsystems

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Carter Wood Blum

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Paul Schrater

December, 2020

Acknowledgements

There are many people who supported me throughout my graduate studies.

I would first like to thank my advisor, Professor Paul Schrater. Thank you for your excellent advice, your incredibly deep well of knowledge and your personal support.

I also owe Professor Maria Gini a huge thank you. I really appreciated you giving me my first strong exposure to research, helping me through tough situations, and advising me along the whole way.

I would also like to acknowledge all of the help and opportunity given to me by Professor Hui Xiong. I relished the opportunity to work with you and the rest of the team at Baidu, especially Dr. Hao Liu, who advised me during my work there. Without the time at Baidu, a large part of this thesis would not have been possible.

Thank you to all of my friends for the great times and support. A special thank you to Suhail Alnahari, for being a great sounding board and partner for trying many ideas.

Finally, I owe it all to the Lord above for being in a position where I'm even able to pursue these interests in goals. I am truly blessed.

Dedication

To Nina Bielinski, for her steadfast optimism.

To Steve & Debbie Blue, my parents, for the foundations of my life.

To Emil Moffa, for teaching me to really think.

Abstract

As electric vehicles have surged in popularity, the problem of ensuring that there is sufficient infrastructure to charge them has attracted large amounts of interest. A key component of this is leveraging existing electric vehicle charging station capacity by intelligently recommending drivers to nearby open stations so they can recharge quickly. Greedily approaching such recommendations can quickly lead to long wait times, so this thesis proposes a method using reinforcement learning to provide recommendations to drivers. While common deep reinforcement learning models fail, exploiting regularities in the state space allows the problem to be decomposed in to smaller problems that can be modeled separately. Experiments demonstrate that this method not only decreases the time before vehicles can start charging by up to 47%, but also that decomposing the problem allows recommendations to be given 2.5x faster than other deep learning based models.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reinforcement learning is an area of machine learning that is quickly garnering interest due to its massive potential applications. However, modern reinforcement learning systems are still data inefficient, difficult to train, and often unstable. This has lead to a significant gap between research and industry. Whereas research in reinforcement learning has progressed quickly, the technology has still had limited success in application.

There are a variety of reasons for this phenomenon, part of which being that deep reinforcement learning models are difficult to debug and diagnose problems in - they often represent a black box. Additionally, the sheer expressivity of these models can allow them to pick up on spurious trends just as easily as they can miss important features of their environment that might seem natural to a human. From this standpoint, it is useful to consider breaking apart models in to separate parts with a similar structure and limited interactions between parts.

This can be helpful from a number of angles. From an interpretability standpoint, increased homogeneity between parts of the computation can decrease the amount of

behaviour that a human designer has to understand. From a computational standpoint, it allows humans to add their knowledge to a network, substantially decreasing the search space of configurations and improving generalizability [1].

The focus of this thesis is to explore and advance the ways that these principles can be applied and conceptualized of in the context of reinforcement learning. We consider a case study in recommending electric vehicle users to charging stations, improving both recommendation quality and speed by exploiting regularities in the state and action space. Specifically, the structure of this thesis is as follows:

- Chapter 1 introduces and contextualizes the goals of this thesis.
- Chapter 2 surveys the relevant literature in reinforcement learning, representation learning, and spatiotemporal recommendations.
- Chapter 3 describes the applied problem solved in this thesis and relates it to reinforcement learning.
- Chapter 4 explains the methods that were used to approach this problem.
- Chapter 5 evaluates this approach, providing results and competitive baselines. It also discusses the implications of the results
- Chapter 7 summarizes the above and identifies future directions.

Chapter 2

Review of Relevant Literature

There has been much research in to ways to improve reinforcement learning and mitigate many its difficulties that were mentioned above. The majority of the work has been done on model-free methods, which attempt to learn a task without modeling its environment, usually via either Q-learning or policy gradient methods. A review of relevant literature in these areas is provided in sections 2.1 and 2.2 below, respectively. Methods relying on modeling the environment will be discussed in section 2.3.

In this thesis, we take particular interest in learning disentangled subsystems that can be applied to reinforcement learning. Disentanglement is often approached from two angles: learning temporally extended policies (hierarchical reinforcement learning) and learning models of data (representation learning). This thesis will take special interest in representation learning, discussed in section 2.4 below. Finally, literature relevant to the problem of charging station recommendation is reviewed in 2.5.

2.1 Value-Based Methods

Value-based methods operate by learning an estimator the expected total rewards that an agent will receive if it takes a given action in a given state. This approach was first codified in to Q-learning in [2], which presented an algorithm for efficiently learning to estimate these rewards using temporal differencing in a Markov Decision Process. A more thorough explanation of this algorithm will be given in 3.2. Although Q-learning was a relatively well-known method, it didn't enjoy widespread attention until after the advent of modern deep learning.

In [3], Mnih et Al. successfully applied neural networks as approximators for the Q-function in Q-learning, enabling autonomous agents to learn to play Atari games from simulation alone, attracting large amounts of interest. In large part due to the success of this paper, neural networks have become the de facto method of Q-function approximation/estimation. The authors also addressed one of the first causes of instability in these methods - deep learning convergence proofs rely on the assumption that the training samples are approximately independent identically distributed. This assumption is violated in reinforcement learning environments, as observations are typically very strongly temporally autocorrelated. This is addressed in [3] and subsequent works by storing observations in a memory buffer, then training the network on random samples from this memory buffer.

This process was further improved in [4], which sampled observations from the memory buffer with probability proportional to the neural network's error in predicting the Q-value. Experimentally, this improved not only the convergence speed, but also performance by forcing the approximator to spend more time on networks that are more 'informative'. Further improvements were made to the stability of these methods by generically altering the structure of the networks used for approximating the Q-function

in [5] and [6].

Another successful method has been to model the distribution of total rewards, not just the expected value. This approach was pursued in [7] [8]. Intuitively, this provides more signal in the learning process of the Q-function, resulting in a more robust estimator. The neural network was also used to induce a distribution in [9], allowing for more efficient exploration. These methods have recently been combined in [10], achieving state-of-the-art performance on Atari and MuJoCo baselines.

2.2 Policy-Gradient Methods

Another approach to learning control tasks is to directly learn a policy for taking actions without necessarily learning estimates of values received from taking specific actions. The first major algorithm to use this approach was REINFORCE [11]. It is assumed that there exists some gradient-based function that gives probabilities of taking each action given a state. Each time an action is chosen, gradient ascent is used to make that action more likely to be chosen in the future, with the size of the update proportional to the reward received from taking that action. A baseline, often called a 'critic', is typically used to cause actions that perform below the baseline to become less likely in the future.

This approach was further popularized in the deep-learning domain by [12], which implemented an asynchronous version of the algorithm, asynchronous advantage actor-critic (A3C), while incorporating some of the strategies found in [6]. A synchronous version of the algorithm (A2C) is also commonly used. These algorithms were further augmented in [13] and [14], which use techniques like sample importance weighting and entropy regularization to further improve the resulting agents. A similar approach was used in [15] to extend these methods from the traditional domain of simulations and

robotics to natural language generation.

In all of the methods discussed above, the space of possible actions is implicitly assumed to be discrete (e.g. the agent goes left OR right). [16] combines ideas from both the value-based and policy gradient methods to tackle problems in a continuous action space (e.g. how many degrees should the agent turn the steering wheel?).

2.3 Model-based Methods

All of the methods described above attempt to directly derive a policy from acting in an environment, without explicitly simulating the environment. Recently, there has been an increased attention in methods that explicitly model their environment and use this information for a number of uses.

In [17] and [18], a model is used to plan possible trajectories. While [17] uses a hard-coded simulation as its model, [18] generalizes this by dynamically learning a model of the world, a method that is popular in many other approaches. While these methods can be computationally expensive, they do have a major benefit in interpretability - it's easy to see what sequences of actions the agent is considering and what the algorithm believes the outcomes of those actions will be.

Besides planning, learned models of the world have been used to improve the sample efficiency of reinforcement learning algorithms. Many algorithms use models to create machine-learned simulations of the environments, allowing them to train the 'model-free' algorithms discussed above inside of these simulations while potentially decreasing the number of interactions needed with the environment [19], [20], [21] [22]. These methods do potentially allow humans to inspect what parts of the environment are better understood, but they do not intrinsically attempt to disentangle the decision making process of the agent itself.

Additionally, models have been used to augment the exploration process, during which the algorithm gathers new observations. Many algorithms attempt to exploit model uncertainty and planning to identify interesting actions to explore [23][24]. These algorithms can again prove to be more sample efficient and stable than standard baselines, but they can still be difficult to interpret.

2.4 Representation Learning

Choosing an effective representation for modeling a problem can substantially improve the learning process, both reducing computational time and increasing training efficiency. This frequently takes the form of implicitly including useful priors in the structure of the model, whether it be through removing connections between variables that are likely to be spurious or by tying parameters representing similar processes together.

One of the earliest and most famous examples of this is that of the convolutional neural network (CNN), first proposed for processing document images by Yann LeCun in 1998 [25]. The functions that can be represented by a convolutional layer are a strict subset of those that are representable by a fully-connected layer, as the convolutional layer effectively removes spatially distant connections and ties the weights of the matrix multiplication applied to each patch together. This general structure has proven very effective at processing spatially based data, ranging from images to weather [26] to traffic [27].

This idea is closely related to that of graph neural networks. Graph neural networks accept input in the form of a graph, a set of nodes and edges. In each layer, the representation of each node is only a function of its own representation in the previous layer and those of its neighbors. Further, these relationships are typically fixed across all edges, again increasing the inductive bias in the model. Graph Neural Networks

have been used in many applications, ranging from recommender systems [28] to drug property discovery [29]. Methods such as GraphSage [30] and Graph Attention Networks [31] can be seen as instances of the more general concept of message-passing networks, which again have seen significant success in pruning the connections between parts of the representation.

Similar strategies have been useful for tasks such as modelling physical systems. Recurrent Independent Mechanisms [32] work by modelling physical entities separately most of the time, only passing messages between them under limited circumstances. The common connecting thread between these representation learning methods and hierarchical reinforcement learning is to simplify arbitrarily complex tasks and data by adding a regularized form that governs the interactions between parts of the problem. This principle is one of the core intuitions guiding the work in this thesis.

2.5 Charging Station Recommendation

The first portion of this thesis focuses on the applying reinforcement learning to the problem of charging station recommendation. While other works have applied reinforcement learning to charging station optimization [33][34][35][36], most of these works have focused on optimizing outcomes once vehicles arrive at a given charging station. [37] developed a model for predicting charging station demand and optimizing station layout for efficient allocation.

Reinforcement learning has been applied to a similar problem - taxi allocation. In [38], a tabular approach to Q-learning is used to recommend where taxis should be moved to in order to best meet future demand. This approach was extremely successful, substantially increasing the profits of Didi Chuxing (滴滴出行), China's largest ride service provider. However, this tabular approach heavily exploits the

interchangeability of vacant taxis in order to be computationally tractible. Due to the heterogeneous nature of charging stations, scaling the algorithm to the problem discussed in this thesis proved computationally infeasible, leading to the use of function approximators discussed in section 4. Similar methods are applied in [39][40][41][42][43].

Another approach to this demand-matching problem is given in [44], which attempts to minimize the divergence between the number of available taxis and the number of queries. Finally, A non-tabular approach has also been pursue in several works, including [45], which implements a capsule network to recommend idle taxi cruising.

Chapter 3

Problem Definition

3.1 Problem Overview

In response to concerns regarding the global climate, electric vehicles (EVs) have experienced a dramatic rise in popularity. From 2015 to 2019, the United States saw an annualized 26.9% increase in electric vehicles each year [46]. During that same period, the growth rate was 27.7% in the European Union and a stunning 61.4% in China. However, at the same time, the number of electric vehicle charging stations has not kept up, growing at a rate of 18.4% [47]. Electric vehicles require the chargers at charging stations to refuel. This, combined with other factors, such as local shortages, has placed greater stress on existing EV charging station infrastructure.

Concurrently, more and more people are using online maps services to navigate and find points of interest (POIs). As a result, recommending users of maps to nearby charging stations to maximize allocative efficiency has become an important issue. Recommending users to charging stations in an effective manner can reduce not only the time the driver spends waiting at a station, but also the time spent driving to said

station.

Therefore, it was desirable to create an intelligent software system to be able to automate recommendations. Such a system's algorithm should be able to receive queries from users looking for nearby charging stations, process information relevant to their query, and return recommended charging stations that are likely to result in the best outcomes. To this extent, the algorithm must be able to balance a number of competing factors. First, the algorithm must balance between minimizing the driver's wait time by recommending them to a station most likely to have open spaces upon arrival of the driver and minimizing the driver's drive time by recommending them to stations that are as close as possible. Second, the algorithm must balance between providing a good experience for the current user and allocating vehicles in such a way that it can also provide good user experiences to future users. Third, the algorithm should balance between providing good results and a quick computation time.

3.1.1 Available Information

With these goals in mind, it is useful to consider what information the algorithm has access to when making its recommendations. The first set of information it has access to pertains to metadata of the user's query. The work for this project was done while at Baidu （百度）, China's dominant search engine and creator of China's most widely used maps app, Baidu Maps (百度地图）. As a result, considerations regarding available data are based on data that was available in this context. The system has access to the local time of the user's query, the user's location (in the form of geo-coordinates), the text of the user's query, and an anonymized user identification number (UID).

Additionally, some general information may be derived from the user's query. The

user's location may be used to look up the local weather including, but not limited to, the type of weather and temperature. The date may be used to look up national holidays and, when paired with the location, local holidays. Finally, some basic information about local traffic may also be derived.

As the majority of drivers in major cities have cell phones, usage of the maps app that the the queries are received from may be used to approximate local traffic information. Using the assumption that users in all parts of the city are equally likely to use their phones, and that users are equally likely to be in a car while using the app in all parts of the city, then the total number of queries received by the app can be used to approximate the density of cars in each area. This information can be very useful for planning, because areas with more active cars are likely to have higher charging station usage, which can be useful in estimating future occupancy and future queries.

However, at the time of implementation no direct data was available from electric vehicle charging stations. As a result, data was scraped from websites of major charging station providers. This data included locations and IDs of charging stations and information about their constituent chargers. At any given time, information was available about the current status of each charger, as well as information about the types of charging that the charger supported (e.g. AC/DC, fast/slow).

Notably, no information was available about how users had responded to previous recommendations from the app or information about their previous trajectories. Additionally, it was not possible to ascertain details about the users' cars, which could be useful in identifying what types of chargers they could use. These limitations represented challenges in accurately modeling the world. More information regarding how they affected the design of the models can be found in section 4.

3.2 Application of Reinforcement Learning

Due to the fact that current recommendations can affect the outcomes of future users, this problem can be framed as a sequential decision making problem. Reinforcement learning (RL) is a useful tool for optimizing outcomes in sequential decision making problems. At a high level, the goal of reinforcement learning is to learn a policy π mapping from states $s \in \mathcal{S}$ to actions $a \in \mathcal{A}$, such that some cumulative reward $R \in \mathbb{R}$ is maximized.

Within the context of this problem, the space of possible states $\mathcal{S}$ is the space of all possible values that can be input for the selected fields in the above section. The space of possible actions $\mathcal{A}$ is simply the list of nearby charging stations that the user may be recommended to. The cumulative reward is less clearly restricted. The goal of this problem is to minimize both the wait and drive time for all users, so, all else being equally, increasing any given user's wait time or drive time should decrease the reward that the agent receives. The specific choice of reward function used will be given in section 4.

At each timestep t, the agent observes a state s_t and chooses an action a_t. Based on the action chosen, the agent receives a reward $r_t \sim R(s, a)$ and then observes a new state at the subsequent timestep s_{t+1}. Traditionally, RL assumes that the environment is a Markov Decision Process (MDP), such that the probability distribution of s_{t+1} is totally defined by the directly preceding state s_t and action a_t such that $s_{t+1} \sim T(s_t, a_t)$. To fit with this restriction, the agent must either have an internal memory in addition to its observations, or the state representation must be specifically chosen to ensure that the environment is a true MDP. The second approach was chosen for its relative simplicity, so information about the system's recent recommendations was appended to the state. A brief analysis of data confirms that this fulfills the Markov property, as

detailed in section 5.

The agent continues choosing actions until the end of an 'episode' after which the environment is assumed to reset. Based on observations that very few queries were made during the late night, most charging stations are vacant at night, except for a few vehicles that charge nightly. Because of this, the recommendations of one day have very little effect on the occupancy at the start of the next day, so episodes were chosen as 24-hour periods resetting at 2 AM. This simplifying assumption allows us to use a large body of methods designed for episodic RL.

One method for maximizing the reward is Q-learning, as discussed in section 2. The Q-value is defined as the expected total cumulative rewards that an agent expects to receive if it takes an action a at state s.

$$Q(s_t, a_t) = E_{r_t \sim R(s_t, a_t),\ s_{t+1} \sim (T(s_t, a_t)} \Big[r_t + \gamma V(s_{t+1}) \Big] \tag{3.1}$$

where $0 \leq \gamma \leq 1$ is a discount factor that weights future rewards less heavily and the value of a state $V(s_t)$ is given by

$$V(s_t) = E_{a_t \sim \pi(s_t)} \Big[Q(s_t, a_t) \Big] \tag{3.2}$$

The goal of Q-learning is then to learn the Q-value of each state-action pair and choose an action corresponding to the highest Q-value at each state with probability 1.

In practice, this is often achieved by using a neural network Q_θ, parameterized by θ, to estimate the Q-values. Namely, the following loss function is minimized:

$$\theta^* = \text{argmin}_\theta \left[\Big(r_t + \gamma V(s_{t+1}) - Q_\theta(s_t, a_t) \Big)^2 \right] \tag{3.3}$$

The Q function is then trained with the temporal differencing algorithm by taking the gradient of the loss with respect to θ and performing gradient descent. If the Q

function estimator has enough representation power, then $Q_{\theta^*} = Q$, i.e. our estimator perfectly models the Q-function. In standard double Deep Q-Networks, it is common to approximate

$$V(s_{t+1}) \approx Q_\phi\left(s_{t+1}, \text{argmax}_{a_{t+1}}\left[Q_\theta(s_{t+1}, a_{t+1})\right]\right) \tag{3.4}$$

There are two things to note here. Firstly, if we assume that $Q_\phi = Q_\theta = Q_{\theta^*}$ and that our policy is to take the action with the highest Q-value with probability 1, then equation 3.4 reduces to equation 3.2. Second, we use different parameters ϕ and θ for the two Q estimators in this equation. This is done to avoid the winner's curse [48], where the action that is chosen by maximizing Q_θ is also likely to be the action which Q_θ most overvalues, biasing V to be higher than it should. By using one network to choose a_{t+1} and another to evaluate it, this method mitigates the effects of the winner's curse. In practice, ϕ is usually a lagging version of θ, which represents a middle ground between addressing the winner's curse and saving computational resources.

Chapter 4

Methodology

Discussion of methodology will be split in to three primary sections:

- *Data Exploration,* which provided insights that guided the rest of the project
- *Simulation Design,* which was used to build a simulation to train the RL agent
- *Model Design,* which ultimately allowed the system to learn to make intelligent recommendations

The project focused on data from 5 major Chinese cities - Beijing, Shanghai, Chongqing, Hangzhou and Tianjin. Of these three cities, 2.5 months of data was collected for Beijing and about 1.5 months of data was collected for each of the others. Information on the number of samples available and the dates of collection is presented in appendix A.5.

4.1 Data Exploration

The first stage in implementation was to identify which of the available variables were likely to be the most useful for the project. This largely consisted of analyzing two pools

of data: those pertinent to traffic patterns (useful for anticipating charger demand) and those related to charger occupancy (useful for modeling charger supply).

As mentioned in section 3.1.1, traffic information was derived by calculating the density of queries to the Baidu maps app in a given location at a given time. Specifically each city was divided in to a grid of cells of each approximately 0.217 square kilometers. Due to considerations regarding occupancy data availability discussed below, traffic information was calculated in 15 minute batches. To represent the traffic density at a given place and time, the number of queries within the corresponding cell during that 15 minute interval was counted.

When plotting traffic patterns by day of week, there was a moderate difference between traffic on weekdays versus weekends. However, there was no discernible difference between individual days of the week. Furthermore, weekend/weekday seasonality aside, traffic patterns did not appear to substantially shift As a result, the agent was provided with a boolean variable corresponding to whether the day it was operating in was a weekday or not.

While there were some notable spatial patterns in the traffic, including hubs outside of the the city center and visually obvious highways, there were no obvious candidates for high-level features that could be extracted. The primary utility of modelling traffic would be to predict trends in queries or predict trends in arrivals. However, counter-intuitively, the number of EV charging station queries recorded in a given area did not strongly correlate with the total number of queries, nor did the number of arrivals at nearby charging stations. Figure 4.1 plots the estimated traffic density on the X axis versus the average number of observed EV charging station queries at each timestep on the y-axis for data from Beijing. Due to this, all information regarding traffic was dropped from both the simulation and the model.

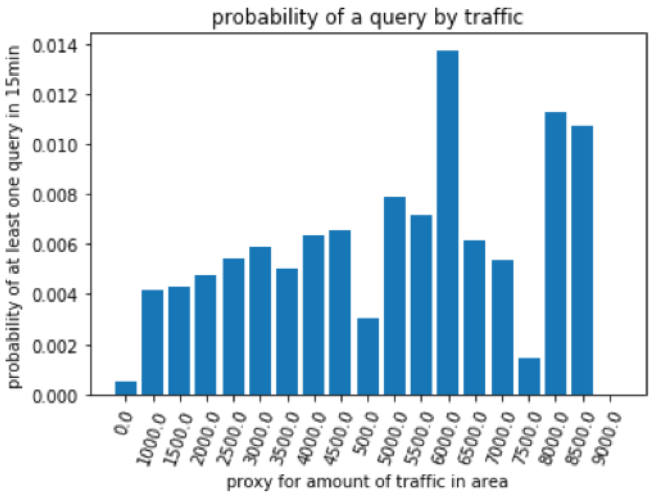

Figure 4.1: The number of queries received in an area doesn't correlate meaningfully with the number of cars in the area. Note: estimates are less accurate on the right-most end of the curve, as there are fewer datapoints.

Despite the impotence of the traffic data, modelling the occupancy of charging stations was still a critical task for identifying when consumers would face long wait times. If a charging station has no available chargers, then every driver who arrives at that station will either have to find another station or wait until there is an unoccupied charger. While there are multiple varieties of charger at each charging station, no information is available to the system regarding the type of electric vehicle that is looking for a place to charge. Additionally, when a charger becomes occupied, there is no way to identify which type of charging functionality is being used at that charger, which might affect charging times. As a result, it is infeasible to match vehicles up with chargers of the corresponding type, so the decision was made to simply model all chargers as a homogeneous group without regard to charger type.

Because there were no available partnerships with electric vehicle charging companies at the time, data was obtained by crawling publicly available webpages. Namely,

two of China's largest electric vehicle charging companies, TeLaiDian (特来电) and the State Grid Corporation of China (国家电网) expose publicly facing websites providing the location of charging stations and the status of those stations' constituent chargers. Some locations only provided the number of available chargers, while other stations provided status codes for each of their chargers - most codes corresponding to 'vacant', 'in-use' and 'out-of-service'. Due to query speed limitations, charger occupancy data was only available at 15 minute intervals.

Upon acquiring the data, several anomalous patterns became apparent from the data that was available regarding specific chargers. First, not all chargers were always listed for each station - some days some of them would be missing. Second, about 20-25% of the chargers were almost constantly marked as full, while 5-10% of them were almost always listed as empty. In cases like these, we assumed an error with the machine's status signal, and such chargers were removed from the dataset. Finally, there were occasionally rare status codes that appeared only once or twice in the dataset. These were marked as out-of-service, as they generally exhibited similar behaviour.

One of the core concerns when representing charging station occupancy is that the representation does not violate the Markov property. There are two core ways that the Markov property might be violated : temporally extended effects might effect when chargers become free and past recommendations will recommend which stations are likely to have new arrivals. Consider the first concern. It's reasonable to assume that the most important variable in predicting when a given charger will become unoccupied is the amount of time the vehicle currently using the station has been charging. One might anticipate that vehicles that have been charging for greater amounts of time are more likely to finish charging soon. Surprisingly enough, the data provided a different

answer.

One way to begin estimating how long a car will be charging at a station is to look at the distribution of charging durations. Interestingly, this distribution closely matched a Geometric distribution, as shown in figure 4.2, ($p < 5e-4$). Notably, there are fewer durations that last for one timestep than would be expected according to a geometric distribution, but this is attributed to the fact that our algorithm may not do a good job at detecting visits with duration 1. This fact is particularly useful, because the Geometric distribution is a memoryless distribution, so knowing how long a car has already been occupying a charger does not substantially aid in predicting how much longer that car will continue doing so. As a result of this, there is no need to keep track of individual cars once they begin charging at a station - simply keeping track of the total number of cars charging is a sufficient statistic. This substantially reduces the amount of information that needs to be collected for the system while also substantially decreasing the size of the input to the system.

4.2 Simulation

These observations made simulating the EV charging station query dynamics much more feasible. Building a simulation is useful because it provides a method for an agent to interact with an environment and see the effects of its actions. In the context of reinforcement learning, the only real alternatives would be off-policy learning and real-world learning. Having the agent learn from scratch in the real world is not an enticing prospect, as one would expect the agent to initially make many mistakes, providing some very bad customer experiences. Off-policy learning involves the agent learning from previously gathered data about the effects of actions. No data about previous recommendations (i.e. actions) is available, so this avenue is also infeasible.

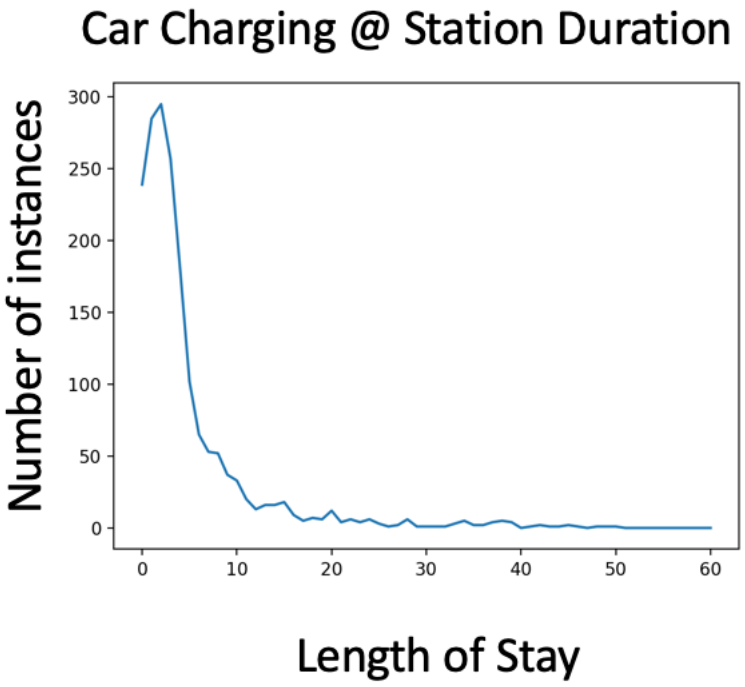

Figure 4.2: Number of times that vehicles were observed staying at a station for a given duration. Computed over the month of July on subset of stations where detailed charger information was consistently available.

When making such a simulation from historical data, there are two obvious strategies that can be followed. The first is to gather aggregate statistics about the historical data, deriving probability distributions for various events, then simulating the events of a day by sampling from these distributions. The second strategy is to randomly select a day from the period in the historical data and simulate the events of that day.

Both methods have advantages and disadvantages. The second method risks overfitting to the historical data, where it only learns how to optimally behave in the exact scenarios that were observed in the historical data. Meanwhile, the first method risks not accurately representing the dynamics of the real world if the probability distributions derived from data do not fully represent the true dynamics of the system. For instance, if the probabilities of two events are modeled as independent when they in fact covary strongly, this could drastically alter the optimal strategy for recommendation away from what is optimal in the real world. Numerous errors arising from gaps between simulation and reality have been documented [49]][50][51] and this difference can lead to arbitrarily poor performance that can be difficult to detect while in the simulation. Another possible source of error in this method is sparse data - as probability distributions are conditioned on more and more variables (to avoid the first problem discussed), the corresponding amount of data that these distributions are derived from becomes correspondingly smaller.

To balance between the concerns listed above, a simulation was built using primarily the first method - sampling from real days that occurred in the data. However, to fight overfitting, an amount of noise was added to these events, as is detailed below. Furthermore, all results listed in the experiments section are computed based on days that are entirely excluded from the training set.

Building a simulation primarily consists of modelling four components:

1. *Queries:* Generating queries that the agent must provide recommendations for
2. *Recommended Arrivals:* Modelling vehicles driving to and arriving at stations after receiving a recommendation from the agent
3. *Other Arrivals:* Vehicles that arrive at stations with no intervention from the agent, occupying spaces
4. *Departures:* Modelling vehicles departing from stations, leaving vacant chargers

Whereas four parts of the simulation contribute to modelling the state, only the second part (recommended arrivals) directly affects the rewards received by the agents. For each of the component descriptions below, assume that some arbitrary day d has been sampled from the set of days that have been recorded in the dataset.

4.2.1 Simulation: Queries

Let $\mathcal{Q}_d$ be the list of queries received on day d. Each query is represented as tuple containing the time $q_t \in \mathbb{N}_0$, coordinates $(q_x, q_y) \in \mathbb{N}_0^2$ and a UID. It was notable that a very high percentage of queries were caused by the same UIDs re-querying the system in quick succession - single users searching multiple times in a row. As such, whenever multiple queries for electric vehicle charging stations were received from the same UID within 15 minutes, only the first such query is included when making the simulation. The UID field is otherwise not used in the simulation.

As noted in section 4.1, both time and space are discretized. q_t is represented by denoting the number of 15 minute time intervals that have passed since 2 AM. The city is divided in to rectangular cells of approximately 0.217 square kilometers in size, with the walls of the rectangles running parallel to lines of latitude and longitude. The grid

has a finite size, ending just outside of city bounds in each direction. q_x and q_y represent the indices of the corresponding grid cells for the geocoordinates of the query.

To simulate the queries, a small amount of noise is added to the values of each query. Noise is sampled from a normal distribution, rounded to the nearest integer, and added to each of $q_t,\ q_x$ and q_y. More specifically

$$\epsilon_t, \epsilon_x, \epsilon_y \sim \mathcal{N}(0, 1)$$

$$\hat{q}_t := q_t + \text{round}(\epsilon_t)$$

$$\hat{q}_x := q_x + \text{round}(\epsilon_x)$$

$$\hat{q}_y := q_y + \text{round}(\epsilon_y)$$

At timestep t, the simulation takes all queries where $\hat{q}_t$ and shuffles them. The first query is then given to the agent as part of the state. When the agent gives a recommendation for the query, the next query is provided and so on until all of the queries for a given timestep have been processed. The agent is not provided information about how many more queries will be received this timestep, as such information would not be available out of simulation.

When each query is given a recommendation, the internal state of the simulation is updated to reflect that the vehicle has received the recommendation. The vehicle's start location is given by $(\hat{q}_x, \hat{q}_y)$.

4.2.2 Simulation: Recommended Arrivals

After a car receives a recommendation from the agent, it is assumed to move directly towards the charging station to which it was recommended. This simplification is made because there is no historical data on how the vehicle drivers responded to previous query results. However, it is not a significant limitation of the simulation - if the

vehicle driver ignores the recommendation, it is effectively as if the agent never gave a recommendation at all. The only real difference is that the agent received information on a particular user's whereabouts and interest in electric vehicle charging stations, which is negated by the fact that they ignored the recommendation and are presumably not much more likely to query the agent in the near future than any other particular user.

Due to the lack of historical trajectory data, it's also difficult to derive an intelligent model for vehicle's motion towards the recommended station. As a result, we use a simple heuristic to estimate the amount of time that it will take the vehicle to reach the station. The number of timesteps it will take to reach its destination is then added to the state (rounded up). After that number of timesteps has expired, the vehicle is marked as having reached its destination station, at which point, it will begin occupying a charger at the station. If there are no chargers at the station, the vehicle re-queries the system to receive a new recommendation. Oftentimes, the system will simply recommend the station that the vehicle is already at, in which case the vehicle is recorded as waiting for one more timestep, at which point it will 're-arrive' at the station and repeat all of the above steps.

There are two components to the reward function: reward from driving time and reward from wait time. The agent receives a fixed negative reward of (-1) for each vehicle that is in transit at each timestep. Over the lifetime of the simulation, this has the effect of providing a negative reward proportional to the total amount of time that recommended cars spend driving. This method of calculating the rewards has two major benefits. First, the rewards at each timestep are a deterministic function of information that the agent has access to - the number of cars that it has recommended recently that are not estimated to have reached their destination. This fulfils the Markov

property, as no information from previous or hidden states is used in the calculation of the rewards. Additionally, it is impossible to avoid receiving these negative rewards - previous versions of the simulation provided negative reward whenever a vehicle began charging at its destination, with the magnitude of the reward proportional to how long that car had driven/waited. Aside from the fact that additional information must be added to the state to make this reward function Markovian, it also resulted in an exploit where the agent would send all vehicles to the other side of the city. This would result in the cars never reaching their destination within the duration of the episode, and thus the agent receiving 0 negative reward, despite obviously not obtaining desirable outcomes. By distributing the negative reward over the course of the vehicle's trip, the agent can only avoid negative rewards by providing good recommendations.

The waiting time reward is calculated in a similar manner. Whenever a vehicle is recommended to a station that it is already at, the agent receives a reward of $(-\lambda)$, where $0 \leq \lambda$. λ is a hyperparameter biasing the agents towards providing low wait times or low driving times. When $\lambda < 1$, the agent would prefer for the vehicles to spend more time waiting than driving, total time being equal. When $\lambda = 1$, the agent should exhibit no preference. In practice, one would expect waiting to be (slightly) preferable to driving, as the vehicle is not expending fuel and the vehicle driver's attention is not being directed towards driving, so correspondingly, one would expect $0 \leq \lambda \leq 1$.

As a result, the reward function at each timestep is given by

$$R(s_t) = -\Big(N_{driving}(s_t) + \lambda N_{waiting}(s_t)\Big) \tag{4.1}$$

Where $N_{driving}$ and $N_{waiting}$ denote the number of vehicles that are estimated to be driving or waiting at the current timestep, respectively. One common concern in systems with negative-only reward is that the agent may attempt to stop the events that cause negative reward. For instance, in a maze with lava where the agent receives

a negative reward for each timestep it is not at the goal state, the agent may instead learn to 'commit suicide' by ending the episode prematurely. In this simulation, such behaviour is not possible, as the agent cannot affect the duration of its episodes, nor can it affect the number of queries it initially receives. It can decrease the total number of queries it receives, but only by recommending vehicles to stations where they will not wait, which is desirable behaviour.

4.2.3 Simulation: Other Arrivals

Naturally, not all cars arriving at a station will have been recommended there by our system. Cars that we had no interaction with are also important parts of the system, as they have a very large effect on the number of available chargers at a station.

Arrivals at a station can be estimated by looking at the charging station data. This is done by recording visits, where a visit is a tuple composed of a station index, an arrival time, and a duration. On each run of the simulation, all of the visits are derived from data for the day that the simulation is based off of.

As not all of the charging station data is from the same source, the same data is not always available. From some data sources, it is possible to see the status of each individual charger in a station at each timestep. In these cases, a visit is recorded as starting whenever a charger is changes its status from 'vacant' to 'in-use'. The time at which this happens is the arrival time, and the station switching back to 'vacant' is the departure time. In the rare situation where a charger is recorded as changing from 'in-use' to 'out-of-service', the timestep at which this occurs is assumed to be the departure time. The duration of a visit is the number of timesteps between the arrival and the departure of the station. In a similar manner to queries, normal noise is added to the arrival times of the visits.

Data from stations where only the total number of available spots is known are calculated in a similar way, using instances where the total number of occupied chargers increases or decreases. Notably, this does not detect scenarios where a vehicle arrives at the same timestep as another one departs, but, from the perspective of estimating the number of available chargers at a station, this has minimal effect.

While most (78%) of the arrivals are not directly affected by the recommendations of our system, it is important to note that the 22% of vehicles arriving represent a non-negligible portion. As these arrivals were recorded in the historical data but not associated with the queries received, adding in arrivals from queries 'on top of' the ones in data would cause the simulation to effectively increase the number of occupied charging spots by 'double counting' the cars that received recommendations. To counteract this, when modelling the the number of other arrivals at each timestep, a random visit is removed for each recommended vehicle arriving at that timestep. This maintains the correct number of total vehicles in the system at any given time.

Modelling the arrivals at any given timestep is then a matter of simply looking up the (perturbed) visits at the current timestep, randomly removing visits for each recommended car that is also arriving. The visits are then shuffled in with the arriving vehicles that received recommendations. Each arrival is takes up a charger if one is available, processed in the shuffled order. If no charger is available, vehicles will query the system for a new recommendation.

4.2.4 Simulation: Departures

As noted above, whenever a recommended vehicle arrives at a station, a random visit from the historical data is removed for that timestep. The duration of that visit is used for the duration of the vehicle that is arriving. Simulating the number of departures

is then simple - after a number of timesteps equal to the duration have elapsed, the number of occupied chargers at the corresponding station is decremented.

Very high-level pseudocode of the simulation is provided in algorithm 1, using python-esque syntax.

Algorithm 1: High-level description of simulation for training agent.

```
% RECOMMENDATIONS keeps track of recommended vehicles that have not yet
  arrived
RECOMMENDATIONS = [ ]
REWARDS = [0] * T_end
for T IN RANGE(T_end) do
    N_WAITING = SIMULATEARRIVALS(RECOMMENDATIONS, STATIONS)
    % recommendations that have arrived are removed from the list
    SIMULATEDEPARTURES()
    % vehicles leave station, freeing up spots
    for QUERY IN QUERIES[T] do
        RECOMMENDATIONS.APPEND(AGENT.RECOMMEND(QUERY))
    end
    N_DRIVING = LEN(RECOMMENDATIONS)
    REWARDS[T] = -N_DRIVING - (λ * N_WAITING)
end
```

4.3 Model

Constructing the model is primarily composed of two parts: representing the relevant information when inputting it in to the model and designing the model itself.

4.3.1 Model: Feature Representation

The relevant state of the system can be split in to three parts.

1. *Global Data:* Weather, Day of the Week, etc.
2. *Station Data:* Location of Station, # of Incoming Cars, # of Open Chargers, etc.
3. *Query Data:* Location of Query

Perhaps the simplest part was the global data, which consists of a vector containing the following information:

- *Weather Type:* Represented as a one-hot vector (see Appendix A.4)
- *Temperature:* Represented as a scalar variable, scaling 0 Celsius through 35 Celsius to 0 to
- *Day of the Week:* Binary variable, representing whether the current day was a weekend or not
- *Holiday:* Binary variable, representing whether the current day was a national holiday or not (see Appendix A.3)

This information, when concatenated, forms $s_t^{(global)}$ at timestep t.

For each station, the data was again concatenated. The system can't recommend the user to stations more than 75 minutes away, which represents 5 timesteps (see 4.3.2 for more details). As a result, the state includes the number of cars that are expected to arrive within each of the next 5 timesteps for each station. This information is available to the agent, as it is simply calculated from the locations of the query and the recommended station. Finally, the number of open chargers and the number of full chargers are included. Both values are normalized by dividing by the 10. As noted in

section 4.1, data about how long the chargers have been occupied is not necessary, so it is omitted.

For each station, let x, y represent the indices of the cell in the traffic grid containing the station. Then x is represented by $\frac{x}{x_{max}}$ and y by $\frac{y}{y_{max}}$, where x_{max} and y_{max} represent the number of grid lines in the x and y direction, respectively. The query data consists only of the location of the query, which is encoded in the same manner.

4.3.2 Model: Architecture

As discussed in the problem definition, the state s is the state of the system, including all of the information discussed above. Due to a large number of stations, the vector containing this information may contain arbitrarily many dimensions. The actions available to the agent are then the stations that the agent may recommend the user to after receiving their query. Each action a can be represented by the station information discussed above. Note that under this formulation, the representations of the actions contain information redundant with the state.

The traditional way to estimate the Q-value in literature is to input the full state in to a deep neural network and output a vector, with each element of the vector associated with the Q-value of a given action. While this approach typically works quite well, it can be quite expensive if the state and/or network are large, and it does not naturally work with scenarios where the action space is changing. These represent substantial limitations for the system, as the size of the network can slow down inference speed and the fact that the output and input vectors are (traditionally) of fixed dimension can make it difficult to incorporate any new charging stations that might be constructed.

Although multiple architectures were experimented with (as referenced in section 5), this section will describe the simplest such algorithm that achieved the best performance.

The key insight in this model is that stations can largely be evaluated in isolation. Furthermore, the process of evaluating the advantage of recommending the user to a given station is largely the same for every single station. The advantage of an action is typically defined as follows:

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{4.2}$$

Which can be thought of as the value of taking action a_t over the average value that the agent would obtain from this state. Therefore,

$$Q(s_t, a_t) = V(s_t) + A(s_t, a_t) \tag{4.3}$$

which follows directly from equation 4.2. Although this initially appears to be non-meaningfully shuffling equations around, it actually can substantially save computation time. $A(s_t, a_t)$ can be efficiently computed on a small fraction of the state. Let s'_t be a vector containing only the global and query data from the above section. In a city of 100s of charging stations, s'_t is multiple orders of magnitude smaller than s_t. We then introduce a new advantage network $\hat{A}$ such that

$$\hat{A}(s'_t, a_t) \approx A(s_t, a_t) \tag{4.4}$$

While the full Q-value is needed during training time due to the temporal differencing algorithm (equation 3.3), only the advantage function is needed for policy evaluation. When the agent is deployed, the value function V never needs to be evaluated, as it is the same for all actions. Only the much cheaper $\hat{A}$ needs to be evaluated in order to provide users with quality recommendations. This is similar to actor-critic systems, where the critic is only needed during training and can be discarded during policy evaluation.

In effect, $\hat{A}$ can be thought of as applying 1D convolution with a kernel size of 1 over each of the stations. This similarity to convolutional networks helps elucidate other benefits of using a uniform $\hat{A}$. It eliminates spurious influences between stations - the occupancy station on the southwest side of a major city is unlikely to have much effect on the quality of a recommendation on the northeast side. Additionally, it ties the weights of all of the stations together, reducing the parameter search space and allowing them to share information. As demonstrated in the experiments section, these changes dramatically improve the performance of the system.

Additionally, it is evident that some recommendations will never be optimal. It isn't worth considering a charging station that is 3 hours away when there are multiple stations less than 15 minutes away. As a result, there is no reason to apply $\hat{A}$ to all of the stations nor give the system the ability to recommend them at all. Therefore, the system only considers stations within a radius r from the query, but not less than k stations. If there are fewer than k stations within radius of the query, the system considers the k nearest stations instead. In the experiments, r was set to 15 grid cells (about 3.25 km)and k was set to 5. This also reduces the search space of policies that the agent must try by blocking off strategies that are obviously poor. [1]

The model is then divided in to to parts V and $\hat{A}$. Both parts of the model are given by feedforward networks of differing size. The V model is a simple feed-forward neural network. The $\hat{A}$ model is nearly identical, except that it is substantially smaller. More details are provided in section 5.2. Both networks naturally output a scalar value which, when taken together, represent the anticipated Q-value for the station that was evaluated.

[1] The restriction on the number of stations considered was implemented after the reward function was changed to distribute rewards. The erroneous strategy observed with the faulty reward system would likely not have been possible to such an egregious extent if the action space were limited.

Both networks were trained using the loss function specified in equation 3.3. The ADAM optimizer [52] was used with an initial learning rate of 3e-4 which was linearly decreased to 3e-6 throughout the course of training. Both networks were trained end-to-end, working in concert with each other.

As is standard in Deep Q-Networks[3], a memory buffer was used with 10000 samples stored at a time. All samples were obtained by running the agent in the simulation. Once the memory buffer had 1000 samples, batches were randomly selected from among them with uniform probability. The samples was stored in the form of tuples (s_t, a_t, r_t, s_{t+1}), as is necessary to compute the loss.

Chapter 5

Experiments

5.1 Baselines

Many maps apps currently use simple heuristic-based approaches for electric vehicle charging station recommendation. In our experiments, we tested three different rule-based models, similar to what one might expect to be implemented in current recommendation systems.

- *Nearby:* The user is always recommended to the station that is estimated to take the least time to travel to. This method should minimize the driving time for all users, at the expense of waiting time.

- *Open:* The user is always recommended to the station with the most open spots. This should minimize waiting time for all users. In practice, this method performs extremely poorly, because it results in extremely high driving times. As a result, it is omitted from the graphical results shown below.

- *Nearest Open:* The user is recommended to the station that is estimated to be

the closest of those that have at least one available charger.

The primary method described in this paper is referred to as the *convolutional (conv.)* model below, as it is equivalent to a one-dimensional convolutional neural network with a 1x1 kernel if all station representations are concatenated along the leading dimension.

An ablation study is also performed on this method. In particular, we examine the following variations of our method.

- *FFDQN:* This method replaces the convolutional advantage function and the value function with a classical dueling DQN architecture [6], using a feedforward neural network. Both the value function and the advantage function have the same structure as the value function in the convolutional method, but here they share the first two layers before splitting off in to two heads. This ablation is intended to test whether or not decomposing the problem provides a tangible advantage to performance.

- *Graph:* A Graph Neural Network (GraphSage) is used to replace the convolutional network. The adjacency matrix is designed such that each station s is adjacent to every other station that could also be a recommended in a query where s is recommended in. Intuitively, this means that each station is adjacent to the stations that it is competing with in the same vicinity. This ablation maintains the decomposed structure of the convolutional model but it provides the ability for stations to pass messages to each other. By comparing it with the convolutional model, we can see if the convolutional model is substantially hampering its performance by omitting such connections.

- *Grouped:* Inspired by [53], this method uses a tiered architecture. We notice that

there are several places where there are multiple charging stations within close proximity to each other. The grid is blocked in to larger 5x5 areas. Summing the capacities and occupancies of the local stations, a representation for the area can be obtained. Areas can then be used in place of stations in the convolutional model, recommending users to an area instead of a specific station. However, once the user enters this area, it then recommends them to an open charging station using the convolutional model, with its choices being limited to the charging stations within that area. The goal of this is to test whether or not there is value in delaying specific recommendations until the user is closer to arriving.

Figure 5.1 provides a graphical overview of the differences between the feedforward, convolutional and graph models.

5.2 Evaluation Details

More data was available for Beijing than any other city, by a wide margin. As a result, the agents are evaluated on simulations derived from Beijing data. Due to the fact that the simulations were based off of real world data, there were irregularities in the information available. Some stations did not have information available at all timesteps, while other stations had erratic tendencies in their recorded data (such as impossibly high occupancies for extended periods of time). While many of the methods above are able to handle a variable number of stations during policy evaluation time, the deep Q-networks all need a constant number of stations during policy training time. To accommodate this, all results below are both trained and evaluated only on the 73 stations in Beijing where consistent data was available throughout the data collection period.

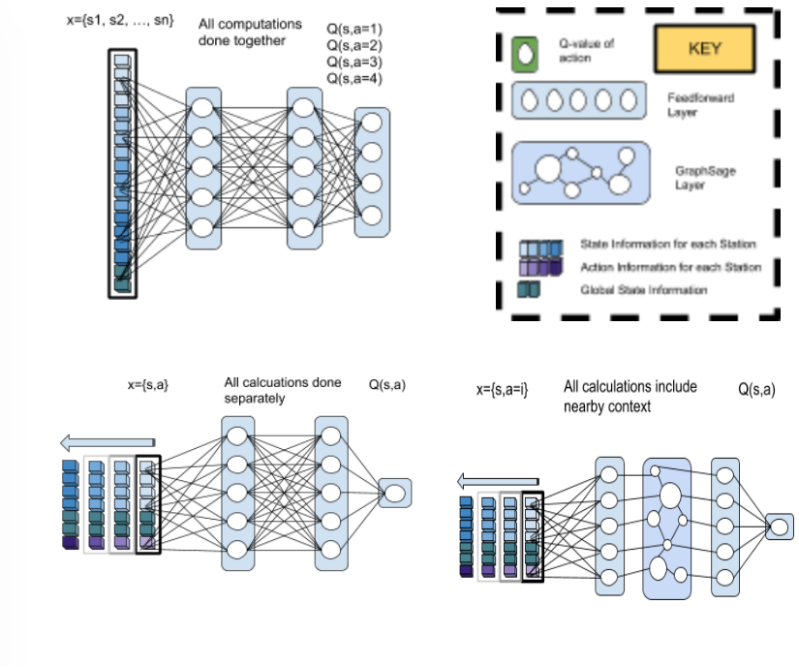

Figure 5.1: Graphical depiction of FFDQN, Conv and Graph models

The city of Beijing was discretized in to a 144 by 126 grid, with each grid cell having an area of roughly 0.217 km^2. More information on the scope of the data used is available in appendix A.5. As described in section 4.2, each run of the simulation was based off of a randomly selected day from the time period for which data was available. To ensure that the models were not overfitting the data collected for these specific dates, seven days were randomly held out from the training data of the simulation.

To give a fair comparison, these same seven days were used to evaluate all of the models reported on. As deep reinforcement learning is notoriously unstable, not all runs of each algorithm achieved a good performance. To minimize the effects of this chance, five different models were trained using each method, and the mean values of their results are reported in the section below 5.2.

All feedforward networks had four layers, with 1024 hidden nodes in each layer and each layer followed by a ReLU activation function. ReLU is defined as follows:

$$ReLU(x) = \max(0, x) \tag{5.1}$$

The layers used for the convolutional network and the GraphSage layers were substantially smaller, containing only 64 hidden nodes.

While training, all networks used an initial learning rate of 3e-4 which exponentially decreased down to 3e-6 over the course of the 500 training epochs. Each training epoch consisted of 100 runs of the simulation. Exploration was handled by using ϵ-greedy exploration. The value of ϵ was linearly decreased from 0.0 to 0.1 over the course of training. ϵ was set to 0 during evaluation, as is standard.

For rewards, the λ value described in section 4.2.2 was 1, equally valuing waiting and driving. The reward discount factor, γ was set to 0.99. Finally, a constant reward of +5 was added whenever a vehicle reached a station and began charging there.

5.3 Results

5.3.1 Recommendation Quality

A plot of the reward that the various methods received during each epoch of training is provided in figure 5.2. The reward received by the heuristic-based agents was trivially constant because those agents were not trained.

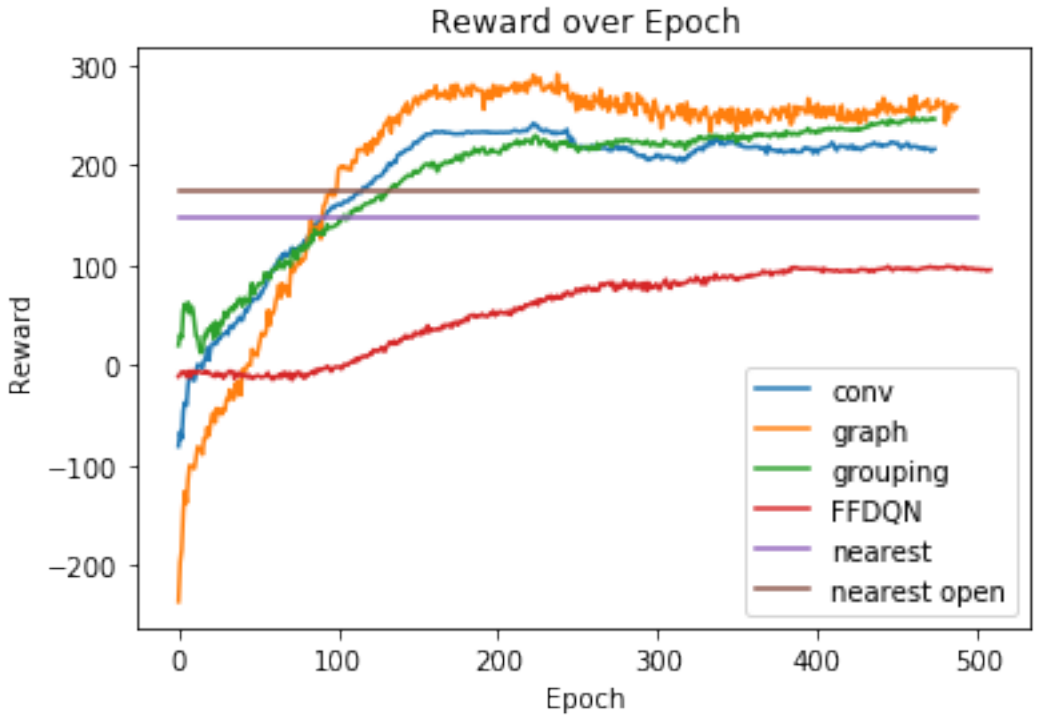

Figure 5.2: Average reward received by agents in test environment while training.

However, figure 5.2 doesn't necessarily provide the clearest picture, as reward is only a surrogate metric for the value we actually care about - minimizing the average time a user has to spend waiting and driving.

The main result of these experiments is summarized in figure 5.3, which displays the relative performance of each method after training is completed. For each method,

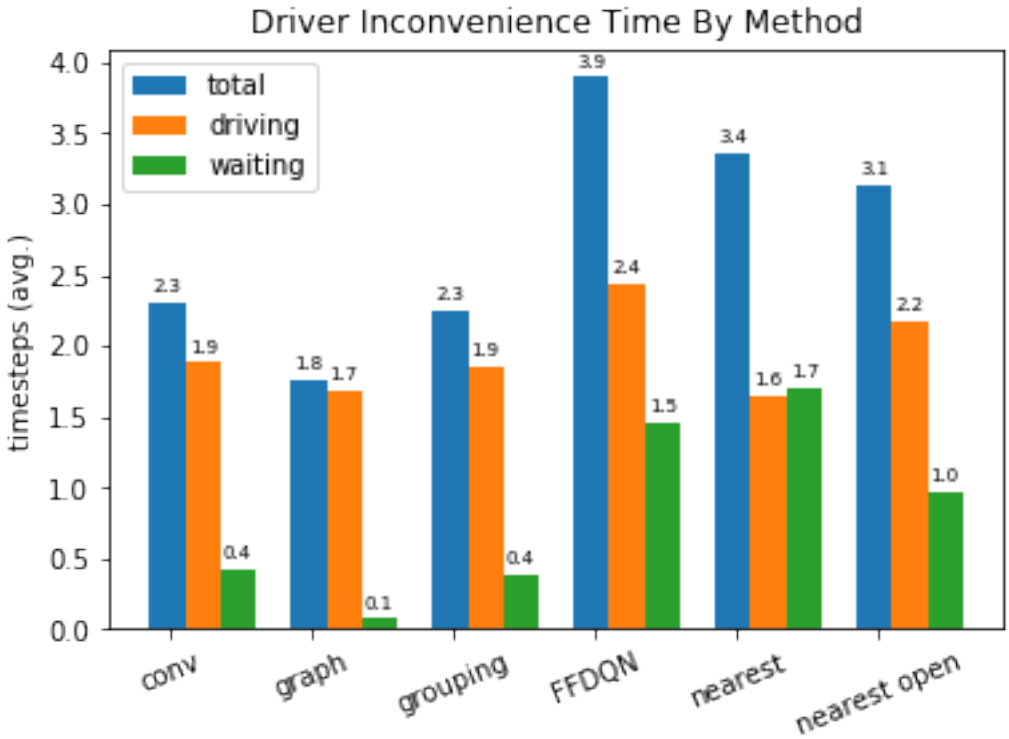

Figure 5.3: Inconvenience to user, broken down by type of inconvenience (driving versus waiting) and recommendation algorithm.

the training checkpoint with the highest test performance is used to evaluate the final performance. In the figure, the sum of the time spent waiting and driving is denoted as 'inconvenience time', i.e. the total amount of time before the user can start charging their electric vehicle after querying the system. Note that the time reported in figure 5.3 is reported in terms of the average number of *timesteps*. Each timestep represents a 15 minute period and if a vehicle is not exactly at its destination, it takes an additional timestep to reach its destination.

In contrast, table 5.1 reports the time take in *minutes*. In this calculation, if a vehicle's distance to its destination is 20% of the distance it is estimated to travel in a single timestep, it is recorded as taking 3 minutes ($\frac{1}{5}$ of a timestep) to reach its destination.

	Graph	**Conv**	**FFDQN**	**Nearest**	**Open**	**Nearest Open**
Reward	**291**	242	97	148	-910	173
Inconvenience Time (m)	**8.80**	11.55	19.46	16.51	80.15	15.62
Wait Time (m)	**0.4**	2.15	7.30	8.50	16.90	4.81
Drive Time (m)	8.40	9.41	12.15	**8.00**	63.25	10.80

Table 5.1: Summary of Recommendation Quality by Method

5.3.2 Computational Speed

Another important metric is the speed with which these methods can be executed. Users will expect a quick response from the server when they query it for information, so the method needs to be able to run quickly.

The average run time for one recommendation using each method is shown in table 5.2. Results are evaluated by averaging the run time of 10,000 runs on an NVIDIA GeForce 1660Ti GPU. Note that the reported time also includes the time to transfer the state to the GPU.

Method	Average Runtime (μs)	Std. Dev. (μs)
Graph	1054	5.44
Conv	301	8.17
FFDQN	810	7.47
Grouped	1152	11.54
Nearest	9.67	0.17
Nearest Open	13.5	0.336
Open	2.01	0.02

Table 5.2: Method Runtimes

5.4 Analysis

Intuitively, 'Nearest Open' outperformed the simpler agent 'Nearest' because it avoided recommendations that would cause long wait times. This intuition is borne out in figure 5.3 and table 5.1, where we observe that 'Nearest Open' cuts wait times compared to 'Nearest', while increasing driving times. However, no method is able to completely avoid wait times, as chargers may become occupied while the recommended car is in transit, causing the recommended car to wait.

One striking result is that the classical feedforward DQN failed to perform well. Its reward did not even approach that of the simple heuristic-based approaches, suggesting it did not even successfully learn to compute distances. In contrast, the rest of the deep learning based approaches all performed competitively, each improving upon the heuristic-based approaches by a substantial margin. One possible explanation for why the FFDQN failed to learn simple heuristics like distance is that it had to learn to compute the distances between the query and 73 different stations separately. To contrast, the convolutional and graphical models only had to learn this once, as all stations 'pass through' the same set of parameters.

Another intriguing insight is that the average rewards received by the graph model

actually decreased after epoch 200. This can be attributed to the aforementioned problem of overfitting - given that the graph model had more representational power, it is possible that it began to overspecialize to the specific days represented in the training data. It is also possible that this is instead an anomaly with the test days - only seven days were used for testing, so it is very possible that they are simply not a representative sample.

Either way, the biggest takeaway is likely that all three non-feedforward methods performed roughly equally on many metrics. This suggests that the specifics on how the network is structured are relatively unimportant. What is important is the disentanglement of the state. The convolutional, graph and grouped models represent three different ways of representing the state, but they all share the same principals - homogeneous pieces of data (stations or grouped stations) are all processed by the same network with the same shared parameters, and connections between pieces of data are limited. This concept was further formalized in [1] as algorithmic alignment, aligning invariants in the structure of the problem with invariants in the structure of the model. Xu et Al show that models that algorithmically align with the problem they are modelling are substantially more sample efficient and generalize better.

The results in this paper reinforce this notion in the realm of reinforcement learning. They demonstrate that algorithmic alignment can be used to substantially improve both learning speed and final performance. In this case, the convolutional model comes with an additional benefit. While it did perform slightly worse than the graph model, it takes a fraction of the time to run. In table 5.2, we note that the convolutional method is at least 2.5x faster than every other method. Not only can this benefit users by giving them faster responses, it can also enable building larger models that could obtain even better results, while still operating quickly and efficiently.

Chapter 6

Conclusion and Discussion

Reinforcement learning is a quickly advancing field with many new improvements and just as many unsolved problems. This thesis discusses one such framework for making in-roads on these problems, by breaking them apart in to pieces. While there are many ways that one could disentangle complex reinforcement learning problems, this thesis highlighted one such way.

This was done through a case study in electric vehicle charging recommendation. These recommendations represent a sequential decision-making problem, because recommendations that are given to users now can affect the availability of chargers for future users. Such recommendations present an interesting challenge in reinforcement learning, due to the high dimensionality of the action space and input space. This feature can make it difficult for reinforcement learning methods to determine what inputs are relevant to what outputs.

However, when the state space is broken apart in to pieces representing individual stations, making recommendations between those stations becomes dramatically easier for the agent. Across a number of different models, it became apparent that this property

had a greater impact on the performance of the model than its other implementation details did. Whereas classical fully-connected networks couldn't even perform as well as simple heuristics, models that broke apart the state and action space not only made recommendations that cut inconvenience time by up to 47% (table 5.1), but they also ran more than 2.5x as quickly as the fully-connected networks (table 5.2).

Looking forward, there is a lot of work to be done in this space. Even within the problem considered in this thesis, there is still room to find ways to improve the performance of the models or strike a balance between the speed of the convolutional models and the recommendation quality of the graph models. This is an exciting time in reinforcement learning, and, as more progress is made, more and more questions are discovered, waiting for answers.

References

[1] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019.

[2] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[5] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[6] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[7] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International Conference on Machine Learning*, pages 449–458. PMLR, 2017.

[8] Will Dabney, Georg Ostrovski, David Silver, and Rémi Munos. Implicit quantile networks for distributional reinforcement learning. In *International conference on machine learning*, pages 1096–1105. PMLR, 2018.

[9] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

[10] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[11] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[12] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[13] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1407–1416. PMLR, 2018.

[14] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

[15] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[16] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[18] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[19] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

[20] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski,

Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

[21] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019.

[22] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

[23] Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak. Planning to explore via self-supervised world models. In *International Conference on Machine Learning*, pages 8583–8592. PMLR, 2020.

[24] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning*, pages 2778–2787. PMLR, 2017.

[25] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[26] Benjamin Klein, Lior Wolf, and Yehuda Afek. A dynamic convolutional layer for short range weather prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4840–4848, 2015.

[27] Haiyang Yu, Zhihai Wu, Shuqin Wang, Yunpeng Wang, and Xiaolei Ma. Spatiotemporal recurrent convolutional networks for traffic prediction in transportation networks. *Sensors*, 17(7):1501, 2017.

[28] Ankit Jain, Isaac Liu, Ankur Sarda, and Piero Molino. Food discovery with uber eats: Using graph learning to power recommendations. `https://eng.uber.com/uber-eats-graph-learning/`, 2019.

[29] Kevin Yang, Kyle Swanson, Wengong Jin, Connor Coley, Philipp Eiden, Hua Gao, Angel Guzman-Perez, Timothy Hopper, Brian Kelley, Miriam Mathea, et al. Analyzing learned molecular representations for property prediction. *Journal of chemical information and modeling*, 59(8):3370–3388, 2019.

[30] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[31] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[32] Anirudh Goyal, Alex Lamb, Jordan Hoffmann, Shagun Sodhani, Sergey Levine, Yoshua Bengio, and Bernhard Schölkopf. Recurrent independent mechanisms. *arXiv preprint arXiv:1909.10893*, 2019.

[33] Konstantina Valogianni, Wolfgang Ketter, and John Collins. Smart charging of electric vehicles using reinforcement learning. In *Workshops at the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[34] Rui Xiong, Jiayi Cao, and Quanqing Yu. Reinforcement learning-based real-time power management for hybrid energy storage system in the plug-in hybrid electric vehicle. *Applied energy*, 211:538–548, 2018.

[35] Yuan Zou, Teng Liu, Dexing Liu, and Fengchun Sun. Reinforcement learning-based real-time energy management for a hybrid tracked vehicle. *Applied energy*, 171:372–382, 2016.

[36] Pu Zhao, Yanzhi Wang, Naehyuck Chang, Qi Zhu, and Xue Lin. A deep reinforcement learning framework for optimizing fuel economy of hybrid electric vehicles. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 196–202. IEEE, 2018.

[37] Anshul Ramachandran, Ashwin Balakrishna, Peter Kundzicz, and Anirudh Neti. Predicting electric vehicle charging station usage: Using machine learning to estimate individual station statistics from physical configurations of charging station networks. *arXiv preprint arXiv:1804.00714*, 2018.

[38] Zhe Xu, Zhixin Li, Qingwen Guan, Dingshui Zhang, Qiang Li, Junxiao Nan, Chunyang Liu, Wei Bian, and Jieping Ye. Large-scale order dispatch in on-demand ride-hailing platforms: A learning and planning approach. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 905–913. ACM, 2018.

[39] Ishan Jindal, Zhiwei Tony Qin, Xuewen Chen, Matthew Nokleby, and Jieping Ye. Optimizing taxi carpool policies via reinforcement learning and spatio-temporal mining. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1417–1426. IEEE, 2018.

[40] Takuma Oda and Carlee Joe-Wong. Movi: A model-free approach to dynamic fleet management. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2708–2716. IEEE, 2018.

[41] Takuma Oda and Yulia Tachibana. Distributed fleet control with maximum entropy deep reinforcement learning. 2018.

[42] Zhenyu Shou, Xuan Di, Jieping Ye, Hongtu Zhu, and Robert Hampshire. Where to find next passengers on e-hailing platforms?-a markov decision process approach. *arXiv preprint arXiv:1905.09906*, 2019.

[43] Abubakr Alabbasi, Arnob Ghosh, and Vaneet Aggarwal. Deeppool: Distributed model-free algorithm for ride-sharing using deep reinforcement learning. *arXiv preprint arXiv:1903.03882*, 2019.

[44] Ming Zhou, Jiarui Jin, Weinan Zhang, Zhiwei Qin, Yan Jiao, Chenxi Wang, Guobin Wu, Yong Yu, and Jieping Ye. Multi-agent reinforcement learning for order-dispatching via order-vehicle distribution matching. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2645–2653. ACM, 2019.

[45] Suining He and Kang G Shin. Spatio-temporal capsule-based reinforcement learning for mobility-on-demand network coordination. In *The World Wide Web Conference*, pages 2806–2813. ACM, 2019.

[46] Alex Kopestinsky. Electric car statistics in the us and abroad. `https://policyadvice.net/insurance/insights/electric-car-statistics/`, 2019.

[47] Ev charging stations statistics. `https://evadoption.com/ev-charging-stations-statistics/`, 2019.

[48] Edward C Capen, Robert V Clapp, William M Campbell, et al. Competitive bidding in high-risk situations. *Journal of petroleum technology*, 23(06):641–653, 1971.

[49] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*, 2019.

[50] Kanishka Rao, Chris Harris, Alex Irpan, Sergey Levine, Julian Ibarz, and Mohi Khansari. Rl-cyclegan: Reinforcement learning aware simulation-to-real. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11157–11166, 2020.

[51] Fereshteh Sadeghi, Alexander Toshev, Eric Jang, and Sergey Levine. Sim2real view invariant visual servoing by recurrent control. *arXiv preprint arXiv:1712.07642*, 2017.

[52] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[53] Tanvi Verma, Pradeep Varakantham, Sarit Kraus, and Hoong Chuin Lau. Augmenting decisions of taxi drivers through reinforcement learning for improving revenues. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*, 2017.

Appendix A

Glossary and Acronyms

A.1 Glossary

- **Atari benchmarks** – A set of 57 common Atari games that are available in OpenAI's gym environment. Due to their diversity and ease-of-use, these are used as the primary baseline in many papers.
- **MuJoCo benchmarks** – Short for '*Mu*lti-*Jo*int dynamics with *Co*ntact.' A simulation engine with a number of common environments frequently used for physics-based control tasks.

A.2 Acronyms

Table A.1: Acronyms

Acronym	Meaning
DQN	**D**eep **Q**-**N**etwork
EV	**E**lectric **V**ehicle
GPU	**G**raphics **P**rocessing **U**nit
FFDQN	**F**eed-**f**orward **D**eep **Q**-**N**etwork
MDP	**M**arkov **D**ecision **P**rocess
ReLU	**Re**ctified **L**inear **U**nit
RL	**R**einforcement **L**earning
POI	**P**oint **o**f **I**nterest
UID	**U**ser **Id**entification Number

A.3 List of Holidays Used

As this work focused on providing charging station recommendations in mainland Chinese cities, the holidays that were included in the input to the system were those designated as national holidays in the People's Republic of China. The list of such holidays is as follows, with their dates for the year 2019 listed:

English Name	**Chinese Name**	**2019 Date(s)**
International New Year's Day	元旦	Jan. 1
Chinese New Year	春节	Feb. 4-10
Tomb-Sweeping Day	清明节	Apr. 5-7
Labour Day	劳动节	May 1-3
Dragon Boat Festival	端午节	June 7
Mid-Autumn Festival	中秋节	Sep 13
National Day	国庆节	Oct. 1-3

Table A.2: National Holidays Considered

A.4 List of Weather Types Differentiated

To represent the current weather in the system, the type of weather was discretized in to a one-hot vector representing the possible values below. Temperature was provided to the system separately.

1. **Snowy**
2. **Rainy**
3. **Smoggy/Misty**
4. **Overcast**
5. **Partly Cloudy**
6. **Sunny**

Table A.3: Weather Types

A.5 Data Availability

The primary limiting factor in building simulations for cities was the availability of data on individual chargers. The table below shows the number of data points on individual chargers that were crawled from the web for each city.

City Name	Data Collection Start	Data Collection End	# Data Points
Beijing	June 26.	Sep. 11	1,188,985
Chongqing	June 26.	Aug. 18	535,567
Shanghai	July 31	Aug 18	432,524
Hangzhou	July 31	Aug 18	325,778
Tianjin	July 31	Aug 13	192,776

Table A.4: Due to instability from the web crawler, not all data was available for the full period of time.